Clase 4 — 14.11.2025

#JAVA

👨‍🏫 *Profesor: Joan Salvador Gordi Ortega*
📘 Programación Multimedia y Dispositivos Móviles
🗓️ Clase 4 — 14/11/2025
🎯 Tema: **Repaso de la Programación Orientada a Objetos (POO) en Java III + Inicio Android**

6️⃣ Interfaces en Java aplicadas a casos reales — Proyecto “Enchufables”

Este ejemplo continúa el repaso profundo de POO y sirve como antesala directa a Android Studio.
El profesor lo utiliza para reforzar:

- **Interfaces como contrato**
- **Polimorfismo transversal**
- **Desacoplamiento**
- **Reflexión**
- **Analogía directa con Android**

6.1 Creación de la interfaz `IEnchufable` — El concepto de contrato

El profesor empieza el ejemplo creando una interfaz muy sencilla:

```
public interface IEnchufable {
    void encender();
    void apagar();
}
```

La gracia de este código no está en “lo poco que tiene”, sino en lo que implica.

Una **interfaz** no es una clase recortada ni una clase a medias: es un **tipo especial** cuyo objetivo es definir un **contrato de comportamiento**. Es decir, describe qué operaciones debe ofrecer un objeto, pero no dice absolutamente nada sobre cómo se implementan internamente.

En `IEnchufable` el contrato es muy claro:
cualquier cosa que quiera considerarse “enchufable” debe saber **encenderse** y **apagarse**. Nada más, pero tampoco nada menos.

El profesor recalca varios detalles importantes:

- En la interfaz **no hay código dentro de los métodos**, solo la firma. Por eso:

  ```
  void encender();
  void apagar();
  ```

  son definiciones “vacías”, sin llaves ni cuerpo. Se está diciendo “existen estos métodos”, pero no “hacen esto”.
- Todas las clases que pongan `implements IEnchufable` están **obligadas** a implementar estos dos métodos. De ahí la metáfora del contrato:

"si firmas, cumples las cláusulas".

- En Java, los métodos de una interfaz son **`public` y `abstract` por defecto**. Por eso escribir:

  ```
  public abstract void encender();
  ```

  sería redundante; el compilador ya asume ambas cosas. Son `abstract` porque no tienen implementación, y `public` porque el contrato tiene que ser accesible desde fuera.

La idea importante es que la interfaz no decide:

- si el aparato tiene pantalla o no,
- si consume más o menos energía,
- si hace ruido o tuesta pan.

Solo fija un **mínimo común denominador** de comportamiento: "algo que se enchufa debe poder encenderse y apagarse".

6.2 Implementación en clases concretas

A partir de aquí, el profesor empieza a "dar vida" al contrato con clases reales:

- `Television`
- `Lavadora`
- `Tostadora`

Cada una representa un aparato concreto, con su propia lógica interna, pero todas cumplen el mismo contrato `IEnchufable`.

Ejemplo de `Television`:

```
public class Television implements IEnchufable {

    @Override
    public void encender() {
        System.out.println("Pantalla encendida mostrando imagen");
    }

    @Override
    public void apagar() {
        System.out.println("Pantalla en negro");
    }
}
```

En el momento en que el profesor escribe:

```
public class Television implements IEnchufable
```

Eclipse se queja inmediatamente con un aviso:
"`Television` debe implementar los métodos heredados de `IEnchufable`".

El IDE ofrece dos soluciones:

- `Add unimplemented methods` → que genere automáticamente los métodos que faltan, vacíos, con el comentario `// TODO...`.

- `Make type 'Television' abstract` → marcar la clase como abstracta para no estar obligada a implementarlos (lo cual aquí **no tiene sentido**, porque queremos una televisión real, no un concepto abstracto).

El profesor selecciona “Add unimplemented methods” y Eclipse genera los esqueletos de:

```
@Override
public void encender() { ... }

@Override
public void apagar() { ... }
```

A partir de ahí, ya solo falta **rellenar la lógica**. En lugar de dejarlo vacío, el profesor escribe mensajes representativos del comportamiento de una televisión:

- Al encender: “Pantalla encendida mostrando imagen”.
- Al apagar: “Pantalla en negro”.

La interfaz no sabía nada de pantallas ni de texto; eso es decisión de la clase concreta. La interfaz solo exigía que existieran los métodos. La **implementación** concreta describe “cómo” se enciende una televisión.

Esto mismo se repetirá con `Lavadora` y `Tostadora`, pero cada una con sus propias acciones:

- La lavadora inicia o detiene un programa de lavado.
- La tostadora empieza o deja de dorar el pan.

El punto clave es que, aunque internamente hagan cosas distintas, externamente todas se ven como algo que se puede `encender()` y `apagar()`.

6.3 Uso del `main()` y primeras instancias

Para probar todo esto, el profesor define la clase `PuntoEntrada` con el clásico método `main`:

```
public static void main(String[] args) {

    Television television1 = new Television();
    Lavadora lavadora1 = new Lavadora();

    TomaDeCorriente toma = new TomaDeCorriente();

    toma.conectar(lavadora1);
    toma.conectar(television1);

    toma.desconectar(lavadora1);
    toma.desconectar(television1);
}
```

Aquí ocurren varias cosas interesantes a nivel de POO y de ejecución:

1. **`main` es el punto de entrada del programa.**
   Cuando se ejecuta la aplicación Java, la JVM empieza exactamente aquí. Todo lo demás que ocurra (instanciación de objetos, llamadas a métodos, impresión de mensajes) parte de este método.

2. **Se crean instancias concretas de clases que implementan la interfaz.**
   `television1` y `lavadora1` son objetos reales, no conceptos abstractos. Cada uno sabe cómo encenderse y apagarse según su propia implementación.
3. **Se crea una instancia de `TomaDeCorriente`.**
   Esta clase será la "encargada" de trabajar con cualquier cosa que se pueda enchufar.
4. Cuando se llama a:

   ```
   toma.conectar(lavadora1);
   toma.conectar(television1);
   ```

   se está diciendo, en la práctica:
   "pásale a la toma de corriente un objeto que cumple el contrato `IEnchufable` y deja que ella decida cómo interactuar con él".

En este momento el profesor introduce la idea clave:

> La variable `lavadora1` es de tipo `Lavadora`, pero **también es de tipo `IEnchufable`**.

Esto es posible porque `Lavadora implements IEnchufable`. Es decir, cualquier instancia de `Lavadora` puede verse, desde fuera, como:

- una `Lavadora` concreta (tipo específico), o
- un `IEnchufable` genérico (tipo interfaz).

Esta dualidad es la base del **polimorfismo por interfaz**:
el código que use `lavadora1` no necesita saber si es una lavadora, una tostadora o una televisión, siempre que lo trate como un `IEnchufable`.

6.4 Problema inicial en `TomaDeCorriente` — Métodos específicos por clase

Para ilustrar por qué las interfaces son tan útiles, el profesor primero enseña una versión "mala" o "naïf" de la clase `TomaDeCorriente`:

```
public class TomaDeCorriente {

    public void conectar(Television television) {
        television.encender();
    }

    public void conectar(Lavadora lavadora) {
        lavadora.encender();
    }

    public void desconectar(Television television) {
        television.apagar();
    }

    public void desconectar(Lavadora lavadora) {
        lavadora.apagar();
    }
}
```

A primera vista, esto “funciona”, pero conceptualmente está lleno de problemas:

1. **Código repetido por todas partes.**
   Las operaciones `conectar` y `desconectar` hacen esencialmente lo mismo para cualquier aparato: llamar a `encender()` o `apagar()`. Sin embargo, tenemos dos métodos para conectar y dos para desconectar, simplemente porque el tipo del parámetro cambia (`Television` vs `Lavadora`).
2. **Crecimiento explosivo del número de métodos.**
   Imagina que añadimos una `Tostadora`, un `Horno`, un `Microondas`…
   Habría que seguir escribiendo:

   ```
   public void conectar(Tostadora tostadora) { ... }
   public void desconectar(Tostadora tostadora) { ... }
   ```

   para cada nuevo tipo. La clase `TomaDeCorriente` crecería sin parar, repitiendo siempre el mismo patrón.
3. **Violación del principio DRY (Don’t Repeat Yourself).**
   Estamos copiando la misma lógica (`objeto.encender()` / `objeto.apagar()`) en múltiples métodos, cambiando solo el tipo del parámetro.
4. **Violación del principio Open/Closed.**
   Cada vez que aparece un nuevo aparato, hay que **modificar** `TomaDeCorriente` para que lo soporte. La clase no está “abierta a extensión y cerrada a modificación”, sino todo lo contrario.
5. **Acoplamiento fuerte entre `TomaDeCorriente` y todas las clases concretas.**
   La toma de corriente “conoce” explícitamente el tipo `Television` y el tipo `Lavadora`. Si mañana cambiamos nombres de clases, paquetes o estructura, tendremos que revisar este código de forma manual.

El profesor señala que, aunque esta solución compila, **no escala** y va en contra de la filosofía de la POO:

> “Con este diseño, cada aparato nuevo implica tocar la toma de corriente. Eso es justo lo que queremos evitar usando interfaces.”

Todo este razonamiento prepara el terreno para la solución limpia: que `TomaDeCorriente` deje de trabajar con tipos concretos (`Television`, `Lavadora`, etc.) y empiece a trabajar con el **contrato común**: `IEnchufable`.

6.5 Solución elegante — Polimorfismo por interfaz

Tras analizar la versión “incorrecta” de `TomaDeCorriente`, el profesor procede a escribir una versión completamente nueva basada en el contrato `IEnchufable`.
Este es el punto donde realmente aparece la potencia de las interfaces dentro de la Programación Orientada a Objetos.

La implementación correcta queda así:

```
public class TomaDeCorriente {

    public void conectar(IEnchufable enchufable) {
        enchufable.encender();
    }

    public void desconectar(IEnchufable enchufable) {
        enchufable.apagar();
```

```
    }
}
```

A primera vista parece casi demasiado simple, pero precisamente esa simplicidad es la evidencia de que el diseño está bien hecho.
¿Qué está pasando aquí? Vamos parte por parte.

1. Eliminamos por completo los métodos duplicados

Antes, había métodos como:

```
public void conectar(Television television) { ... }
public void conectar(Lavadora lavadora) { ... }
```

Cada uno hacía lo mismo: ejecutar `encender()`.
El problema era puramente sintáctico: como el parámetro era distinto (`Television` vs `Lavadora`), Java los trataba como métodos diferentes.

Con interfaces, este problema desaparece:

```
public void conectar(IEnchufable enchufable)
```

Significa: **"pásame cualquier cosa que cumpla el contrato IEnchufable"**.
No necesito saber si es una lavadora, una tostadora, una televisión, una cafetera, un robot aspirador o un secador de pelo.

2. El método deja de depender de clases concretas

La clase no está acoplada a:

- `Television`
- `Lavadora`
- `Tostadora`

ni a ninguna otra futura.

`TomaDeCorriente` ahora solo depende **del contrato**, no de las implementaciones.
Esta es la esencia del diseño desacoplado.

3. El polimorfismo hace el resto del trabajo

Cuando dentro del método se ejecuta:

```
enchufable.encender();
```

Java decide dinámicamente cuál implementación concreta debe usar:

- Si el objeto es una `Television`, ejecutará el código de `Television.encender()`.
- Si es una `Lavadora`, ejecutará `Lavadora.encender()`.
- Si es una `Tostadora`, ejecutará `Tostadora.encender()`.

El profesor insiste muchísimo en esto:

> "El polimorfismo evita que tú tengas que programar un `if` gigante con:
> `si es tele haz esto, si es lavadora haz esto otro…`
> La decisión la hace Java en tiempo de ejecución."

4. La clase ya es escalable por diseño

El profesor explica que esta versión de `TomaDeCorriente`:

- no crece,
- no se modifica,
- no se rompe si se añaden nuevos aparatos.

Es un ejemplo perfecto del Principio Abierto-Cerrado (OCP):

- *Abierta a extensión* → Permite añadir nuevos dispositivos.
- *Cerrada a modificación* → No hay que tocar su código.

Por eso él dice:

> "Gracias a las interfaces, diseñamos sistemas escalables.
> Puedes enchufar cualquier aparato que cumpla el contrato sin tocar el resto del código."

6.6 Añadiendo Tostadora sin tocar la TomaDeCorriente

Para demostrarlo, el profesor añade una clase nueva:

```
public class Tostadora implements IEnchufable {

    @Override
    public void encender() {
        System.out.println("Empieza a dorar pan");
    }

    @Override
    public void apagar() {
        System.out.println("Deja el pan como esté");
    }
}
```

Observa algo esencial:

- No hemos modificado **ni una sola línea** de `TomaDeCorriente`.
- No se han añadido métodos nuevos para "tostadora".
- Todo funciona automáticamente.

Ahora en `main()` podemos hacer:

```
Tostadora tostadora1 = new Tostadora();
toma.conectar(tostadora1);
toma.desconectar(tostadora1);
```

La salida muestra los mensajes específicos de cada aparato:

- Lavadora → "Iniciar programa de lavado" / "Detener programa…"
- Televisión → "Pantalla encendida…" / "Pantalla en negro"
- Tostadora → "Empieza a dorar pan" / "Deja el pan como esté"

Y todo ello usando **el mismo método**:

```
conectar(IEnchufable e)
desconectar(IEnchufable e)
```

Eso es polimorfismo puro.

El profesor remarca:

> "Si un aparato implementa IEnchufable, entonces puede ser conectado y desconectado.
> No me importa cuál sea su marca, modelo o tipo: cumple el contrato."

6.7 Reflexión: usando `.getClass()` para identificar objetos

Después, el profesor introduce un concepto mucho más avanzado: **la reflexión**, que consiste en que un programa puede inspeccionarse a sí mismo en tiempo de ejecución.

Ejemplo que muestra:

```
System.out.println(enchufable.getClass().getTypeName());
```

Si pasamos distintos objetos al método, la salida será:

```
class Lavadora
class Television
class Tostadora
```

Esto permite **descubrir el tipo real** de un objeto sin necesidad de conocerlo de antemano.
Es lo que él conecta con los frameworks:

- Java usa reflexión interna para crear objetos dinámicamente.
- Librerías como Spring, Hibernate, Jakarta, Jackson, Retrofit, etc. usan `getClass()`, `getMethods()` y `getFields()` para automatizar tareas sin que tú programes todo a mano.

> "Imagina que tienes cien clases enchufables.
> Con reflexión y un buen diseño puedes descubrirlas, cargarlas, aplicarles lógica y trabajar con ellas sin escribir código específico para cada una."

El profesor destaca que Android también utiliza reflexión:

- al inflar layouts,
- al cargar Activities,
- al instanciar fragments,
- al generar vistas dinámicas.

6.8 Analogía directa con Android

Esta parte es fundamental.
El profesor une los dos mundos: el ejemplo doméstico de enchufar aparatos y la arquitectura real de Android.

Tabla comparativa:

Concepto en Java	Equivalencia en Android
`IEnchufable`	`OnClickListener`, `Runnable`, `TextWatcher`
`encender()`	`onClick(View v)`
`TomaDeCorriente.conectar()`	`button.setOnClickListener(...)`
Clases concretas (`Television`, `Lavadora`, `Tostadora`)	`Activity`, `Fragment`, clases anónimas
`.getClass()`	Reflexión interna de Android para instanciar vistas, inflar layouts, etc.

Ejemplo de Android:

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // acción
    }
});
```

¿Por qué es equivalente?

- El botón no sabe si quien implementa el listener es una Activity, un Fragment, una clase separada o una clase anónima.
- Solo sabe que implementa `OnClickListener`.
- Por tanto, puede llamar a su método `onClick`.

Exactamente igual que:

```
toma.conectar(IEnchufable e);
```

La toma de corriente no quiere saber qué aparato está recibiendo.
Solo quiere una garantía: **que pueda encenderse y apagarse**.

El profesor concluye diciendo:

> “Android funciona así: no le interesan las clases concretas.
> Solo le interesa que implementen la interfaz correcta.
> Igual que nuestra toma de corriente.”

Si quieres, puedo seguir con:

- **La clase 5 (primer proyecto Android Studio)**
- o ampliar aún más este apartado, incluyendo diagramas UML, explicación del LSP, o ejemplos más avanzados.

6.9 Conclusión del profesor

El profesor cierra la clase con tres ideas fundamentales:

1) Las interfaces permiten agrupar objetos por comportamiento

No por herencia, no por estructura.

2) El polimorfismo por interfaz es la base del diseño desacoplado

Permite escribir métodos genéricos como:

```
conectar(IEnchufable e)
```

y que funcionen con cualquier clase futura.

3) Android funciona exactamente así

Listeners, callbacks, adaptadores, servicios…
Todo está basado en la misma idea que hoy hemos programado.
